CSC 427: Data Structures and Algorithm Analysis

Fall 2004

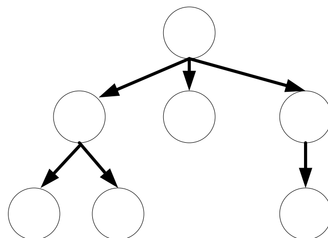
Trees and nonlinear structures

- tree structure, root, leaves
- recursive tree algorithms: counting, searching, traversal
- divide & conquer
- binary search trees, efficiency
- AVL trees

Tree

a tree is a nonlinear data structure consisting of nodes (structures containing data) and edges (connections between nodes), such that:

- one node, the *root,* has no *parent* (node connected from above)
- every other node has exactly one parent node
- there is a unique path from the root to each node (i.e., the tree is connected and there are no cycles)

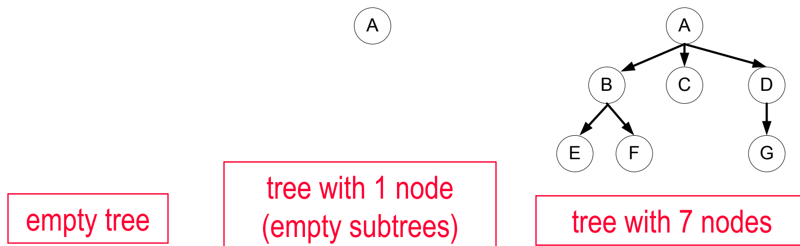


nodes that have no children (nodes connected below them) are known as *leaves*

Recursive definition of a tree

trees are naturally recursive data structures:

- the empty tree (with no nodes) is a tree
- a node containing data and with subtrees connected below is a tree



empty tree

tree with 1 node (empty subtrees)

tree with 7 nodes

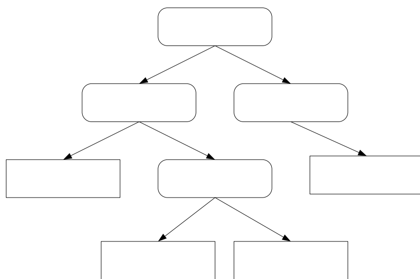
a tree where each node has at most 2 subtrees (children) is a *binary* tree

Trees in CS

trees are fundamental data structures in computer science

example: file structure

- an OS will maintain a directory/file hierarchy as a tree structure
- files are stored as leaves; directories are stored as internal (non-leaf) nodes



descending down the hierarchy to a subdirectory
⇳
traversing an edge down to a child node

DISCLAIMER: directories contain links back to their parent directories (e.g., `..`), so not strictly a tree

Recursively listing files

to traverse an arbitrary directory structure, need recursion

to list a file system object (either a directory or file):

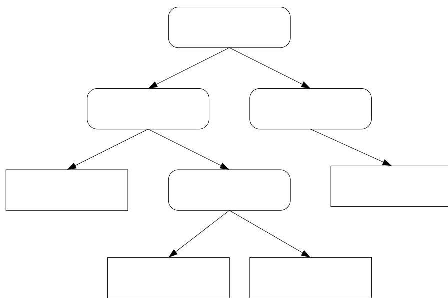
1. print the name of the current object
2. if the object is a directory, then
   a. recursively list each file system object in the directory

in pseudocode:

```
void ListAll(const FileSystemObj & current)
{
   Display(current);
   if (isDirectory(current)) {
     for (FileSystemObj::iterator iter = current.begin(); iter != current.end(); iter++) {
       ListAll(*iter);
     }
   }
}
```


Recursively listing files

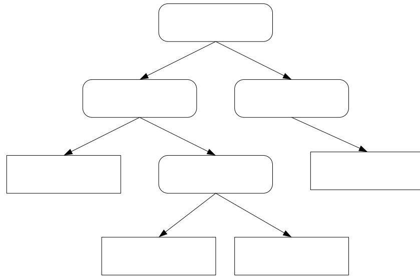
```
void ListAll(const FileSystemObj & current)
{
   Display(current);
   if (isDirectory(current)) {
     for (FileSystemObj::iterator iter = current.begin(); iter != current.end(); iter++) {
       ListAll(*iter);
     }
   }
}
```



this function performs a *pre-order traversal*: prints the root first, then the subtrees

UNIX `du` command

in UNIX, the du command list the size of all files and directories



from the `~davereed` directory:

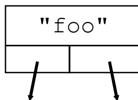
```
unix> du -a
2  ./public_html/index.html
3  ./public_html/Images/reed.jpg
3  ./public_html/Images/logo.gif
7  ./public_html/Images
10 ./public_html
1  ./mail/dead.letter
2  ./mail
13 .
```

```
int du(const FileSystemObj & current)
{
   int size = current.BlockSize();
   if (isDirectory(current)) {
     for (FileSystemObj::iterator iter = current.begin(); iter != current.end(); iter++) {
       size += du(*iter);
     }
   }
   Display(size, current);
   return size;
}
```

this function performs a *post-order traversal*: prints the subtrees first, then the root

Implementing binary trees

to implement binary trees, we need a node that can store a data value & pointers to two child nodes (RECURSIVE!)



NOTE: exact same structure as with doubly-linked list, only left/right instead of previous/next

```
template <class Data> class BinaryNode
{
  public:
    BinaryNode(const Data & val, BinaryNode<Data> * left = NULL,
                                 BinaryNode<Data> * right = NULL)
    {
          data = val;
          leftPtr = left;
          rightPtr = right;
    }

    void setData(const Data & newValue) { data = newValue; }

    void setLeft(BinaryNode<Data> * left) { leftPtr = left; }

    void setRight(BinaryNode<Data> * right) { rightPtr = right; }

    Data & getData() { return data; }

    BinaryNode<Data> * & getLeft() { return leftPtr; }

    BinaryNode<Data> * & getRight() { return rightPtr; }

  private:
    Data data;
    BinaryNode<Data> * leftPtr;
    BinaryNode<Data> * rightPtr;
};
```

Counting nodes in a tree

due to their recursive nature, trees are naturally handled recursively

to count the number of nodes in a binary tree:

BASE CASE: if the tree is empty, number of nodes is 0

RECURSIVE: otherwise, number of nodes is
(# nodes in left subtree) + (# nodes in right subtree) + 1 for the root

```
template <class TYPE> int NumNodes(BinaryNode<TYPE> * root)
{
    if (root == NULL) {
          return 0;
    }
    else {
          return NumNodes(root->getLeft()) + NumNodes(root->getRight()) + 1;
    }
}
```


Searching a tree

to search for a particular item in a binary tree:

BASE CASE: if the tree is empty, the item is not found
BASE CASE: otherwise, if the item is at the root, then found

RECURSIVE: otherwise, search the left and then right subtrees

```
template <class TYPE> bool IsStored(const TYPE & value, BinaryNode<TYPE> * root)
{
    return (root != NULL && (root->getData() == value ||
                            IsStored(value, root->getLeft()) ||
                            IsStored(value, root->getRight())));
}
```

Traversing a tree: preorder

there are numerous patterns that can be used to traverse the entire tree

pre-order traversal:

BASE CASE: if the tree is empty, then nothing to print

RECURSIVE: print the root, then recursively traverse the left and right subtrees

```
template <class TYPE> void PreOrder(BinaryNode<TYPE> * root)
{
    if (root != NULL) {
          cout << root->getData() << endl;
          PreOrder(root->getLeft());
          PreOrder(root->getRight());
    }
}
```


Traversing a tree: inorder & postorder

in-order traversal:

BASE CASE: if the tree is empty, then nothing to print

RECURSIVE: recursively traverse left subtree, then display root, then right subtree

```
template <class TYPE> void InOrder(BinaryNode<TYPE> * root)
{
    if (root != NULL) {
          InOrder(root->getLeft());
          cout << root->getData() << endl;
          InOrder(root->getRight());
    }
}
```

post-order traversal:

BASE CASE: if the tree is empty, then nothing to print

RECURSIVE: recursively traverse left subtree, then right subtree, then display root

```
template <class TYPE> void PostOrder(BinaryNode<TYPE> * root)
{
    if (root != NULL) {
          PostOrder(root->getLeft());
          PostOrder(root->getRight());
          cout << root->getData() << endl;
    }
}
```

Exercises

```
template <class TYPE> int NumOccur(const TYPE & value, BinaryNode<TYPE> * root)
// Returns: the number of times value occurs in the tree with specified root
{


}
```

```
int Sum(BinaryNode<int> * root)
// Returns: the sum of all the int values stored in the tree with specified root
{


}
```

```
template <class TYPE> int Height(BinaryNode<TYPE> * root)
// Returns: # of nodes in the longest path from root to leaf in the tree
{


}
```


Divide & Conquer algorithms

recursive algorithms such as tree traversals and searches are known as *divide & conquer algorithms*

the divide & conquer approach tackles a complex problem by breaking into smaller pieces, solving each piece, and combining into an overall solution

- e.g., to count number of nodes in a binary tree, break into counting the nodes in each subtree (which are smaller), then adding the results + 1

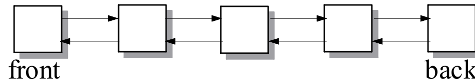
divide & conquer is applicable when a problem can naturally be divided into independent pieces

- e.g., merge sort performed divide & conquer – divided list into half, conquered (sorted) each half, then merged the results

other examples of a divide & conquer approach we have seen before?

Searching linked lists

recall: a (linear) linked list only provides sequential access → O(N) searches



it is possible to obtain O(log N) searches using a tree structure

in order to perform binary search efficiently, must be able to

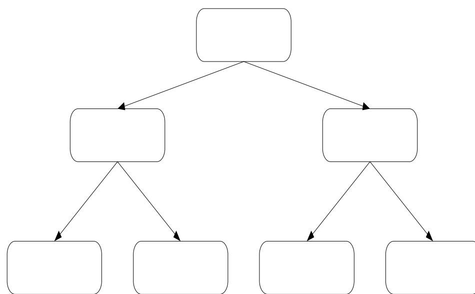
- access the middle element of the list in O(1)
- divide the list into halves in O(1) and recurse

HOW CAN WE GET THIS FUNCTIONALITY FROM A TREE?

Binary search trees

a *binary search tree* is a binary tree in which, for every node:

- the item stored at the node is ≥ all items stored in the left subtree
- the item stored at the node is < all items stored in the right subtree



in a (balanced) binary search tree:

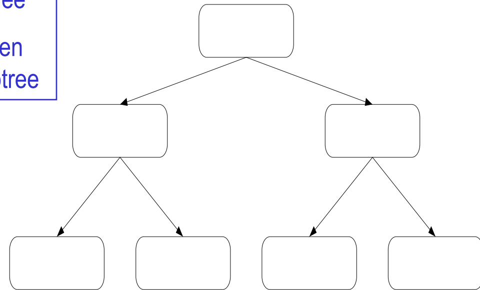
- middle element = root
- 1st half of list = left subtree
- 2nd half of list = right subtree

furthermore, these properties hold for each subtree

Binary search in BSTs

to search a binary search tree:

1. if the tree is empty, NOT FOUND
2. if desired item is at root, FOUND
3. if desired item < item at root, then recursively search the left subtree
4. if desired item > item at root, then recursively search the right subtree



Search efficiency

how efficient is binary search on a BST

- in the best case?

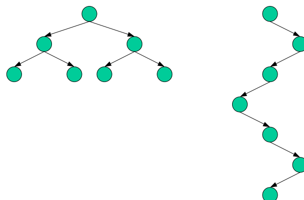
  O(1) if desired item is at the root

- in the worst case?

  O(height of the tree) if item is leaf on the longest path from the root

in order to optimize worst-case behavior, want a (relatively) balanced tree

- otherwise, don't get binary reduction
- e.g., consider two trees, each with 7 nodes



How deep is a balanced tree?

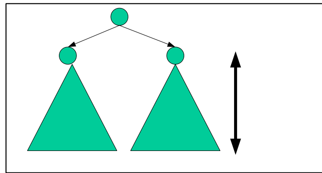
THEOREM: A binary tree with height H can store up to 2^H-1 nodes.

Proof (by induction):

BASE CASES: when H = 0, 2^0 - 1 = 0 nodes ✓
when H = 1, 2^1 - 1 = 1 node ✓

HYPOTHESIS: assume a tree with height H-1 can store up to 2^{H-1} -1 nodes

INDUCTIVE STEP: a tree with height H has a root and subtrees with height up to H-1



by our hypothesis, T1 and T2 can each store 2^{H-1} -1 nodes, so tree with height H can store up to

$1 + (2^{H-1} -1) + (2^{H-1} -1) =$

$2^{H-1} + 2^{H-1} -1 =$

2^H -1 nodes ✓

equivalently: N nodes can be stored in a binary tree of height $\lceil \log_2(N+1) \rceil$

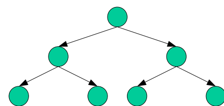
Search efficiency (cont.)

so, in a balanced binary search tree, searching is O(log N)

N nodes → height of ⌈log2(N+1)⌉ → in worst case, have to traverse ⌈log2(N+1)⌉ nodes

what about the average-case efficiency of searching a binary search tree?

- assume that a search for each item in the tree is equally likely
- take the cost of searching for each item and average those costs



costs of search

1

2 + 2

3 + 3 + 3 + 3

→ 17/7 → 2.42

height
H-1

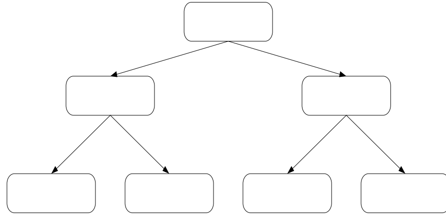
define the *weight* of a tree to be the sum of all node depths (root = 1, …)

average cost of searching a tree = weight of tree / number of nodes in tree

Binary search tree operations

we already saw that *searching* for an item is relatively straightforward

what about *inserting* into a binary search tree?



<u>inserting into a BST</u>

1. traverse edges as in a search
2. when you reach a leaf, add the new node below it

PROBLEM: random insertions do not guarantee balance

- e.g., suppose you started with an empty tree, added words in alphabetical order

with repeated insertions, can degenerate so that height is O(N)

- specialized algorithms exist to maintain balance & ensure O(log N) height →**LATER**
- or take your chances: on average, N random insertions yield O(log N) height →**HW5**

"phillies"

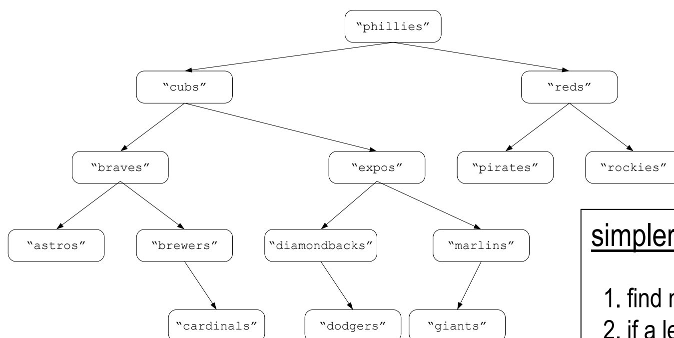
"reds"

Removing an item

tes" "roc

we could define an algorithm that finds the desired node and removes it

- tricky, since removing from the middle of a tree means rerouting pointers
- have to maintain BST ordering property



<u>simpler solution</u>

1. find node (as in search)
2. if a leaf, simply remove it
3. if no left subtree, reroute parent pointer to right subtree
4. otherwise, replace current value with largest value in left subtree

BinarySearchTree class

we can encaspulate all of these operations into a BinarySearchTree class

- since most operations are recursive, will need public "fronts" and private "workers" that take the root pointer as parameter
- also, since the class utilizes dynamic memory, need to define a copy constructor, destructor, and operator=

```
template <class Item>
class BinarySearchTree
{
  public:
    BinarySearchTree() { … }
    BinarySearchTree(const BinarySearchTree<Item> & tree) { … }

    ~BinarySearchTree() { . . . }
    BinarySearchTree & operator=(const BinarySearchTree<Item> & tree) { … }

    bool isStored(const Item & desiredItem) { … }
    void insert(const Item & desiredItem) { … }
    void remove(const Item & desiredItem) { … }
    void display() { … }

  private:
    BinaryNode<Item> * root;

    // PRIVATE MEMBER FUNCTIONS
};
```


BinarySearchTree class (cont.)

```
    BinarySearchTree()
    // default constructor -- initializes empty tree
    {
        root = NULL;
    }


    BinarySearchTree(const BinarySearchTree<Item> & tree)
    // copy constructor -- initializes to be a copy of tree
    // Assumes : Item type supports assignment
    {
        root = copyTree(tree.root);
    }
```

```
    BinaryNode<Item> * copyTree(BinaryNode<Item> * rootPtr)
    // PRIVATE: returns pointer to copy of tree w/ rootPtr
    {
        if (rootPtr == NULL) {
            return NULL;
        }
        else {
            return new BinaryNode<Item>(rootPtr->GetData(),
                                        CopyTree(rootPtr->getLeft()),
                                        CopyTree(rootPtr->getRight()));
        }
    }
```

BinarySearchTree class (cont.)

```
~BinarySearchTree()
// destructor -- frees dynamically allocated storage
{
    deleteTree(root);
}


BinarySearchTree & operator=(const BinarySearchTree<Item> & tree)
// assignment operator -- assumes Item type supports assignment
{
    if (this != &tree)      // don't assign to self!
    {
        deleteTree(root);
        root = copyTree(tree.root);
    }
    return *this;
}
```

```
void deleteTree(BinaryNode<Item> * rootPtr)
// PRIVATE: deletes contents of tree w/ rootPtr
{
    if (rootPtr != NULL) {
        DeleteTree(rootPtr->getLeft());
        DeleteTree(rootPtr->getRight());
        delete rootPtr;
    }
}
```


BinarySearchTree class (cont.)

```
bool isStored(const Item & desiredItem) const
// Assumes : Item supports relational operators (==, <, <=, ...)
// Returns: true if desiredItem is stored in the tree, else false
{
    return searchTree(root, desiredItem);
}
```

```
bool searchTree(BinaryNode<Item> * rootPtr, const Item & desired) const
// PRIVATE: returns true if desired in the tree w/ rootPtr
{
    if (rootPtr == NULL) {
        return false;
    }
    else if (desired == rootPtr->getData()) {
        return true;
    }
    else if (desired < rootPtr->getData()) {
        return searchTree(rootPtr->getLeft(), desired);
    }
    else {
        return searchTree(rootPtr->getRight(), desired);
    }
}
```

BinarySearchTree class (cont.)

```
void insert(const Item & desiredItem)
// Assumes : Item supports assignment, relational operators
// Results: desiredItem is inserted into the correct location
{
    insertIntoTree(root, desiredItem);
}
```

```
void insertIntoTree(BinaryNode<Item> * & rootPtr, const Item & desired)
// PRIVATE: inserts desired into the tree w/ rootPtr
{
    if (rootPtr == NULL) {
        rootPtr = new BinaryNode<Item>(desired, NULL, NULL);
    }
    else if (rootPtr->getData() >= desired) {
        insertIntoTree(rootPtr->getLeft(), desired);
    }
    else {
        insertIntoTree(rootPtr->getRight(), desired);
    }
}
```


BinarySearchTree class (cont.)

```
void remove(const Item & desiredItem)
// Assumes: desired is stored in the tree, Item supports relational ops
// Results: first occurrence of desiredItem is removed from tree
{
    removeFromTree(root, desiredItem);
}
```

```
void removeFromTree(BinaryNode<Item> * & rootPtr, const Item & desired)
// PRIVATE: removes desired from tree w/ rootPtr
{
    if (desired == rootPtr->getData()) {
        if (rootPtr->getLeft() == NULL) {
            BinaryNode<Item> * temp = rootPtr;
            rootPtr = rootPtr->getRight();
            delete temp;
        }
        else {
            rootPtr->setData(findLargest(rootPtr->getLeft()));
            removeFromTree(rootPtr->getLeft(), rootPtr->getData());
        }
    }
    else if (desired < rootPtr->getData()) {
        removeFromTree(rootPtr->getLeft(), desired);
    }
    else {
        removeFromTree(rootPtr->getRight(), desired);
    }
}
```

BinarySearchTree class (cont.)

```
void display(ostream & ostr = cout) const
// Assumes : Item supports "<<" operator
// Results: displays contents of tree via inorder traversal
{
    inOrder(root, ostr);
}
```

```
static Item findLargest(BinaryNode<Item> * rootPtr)
// PRIVATE: returns largest value in tree w/ rootPtr (assumes rootPtr != NULL)
{
    while (rootPtr->getRight() != NULL) {
        rootPtr = rootPtr->getRight();
    }
    return rootPtr->getData();
}

void inOrder(BinaryNode<Item> * rootPtr, ostream & ostr) const
// PRIVATE: displays contents of the tree w/ rootPtr
{
    if (rootPtr != NULL) {
        inOrder(rootPtr->getLeft(), ostr);
        ostr << rootPtr->getData() << endl;
        inOrder(rootPtr->getRight(), ostr);
    }
}
```


Utilizing BinarySearchTree

```
class Dictionary
{
  public:
    Dictionary();
    void read(istream & istr = cin);
    void write(ostream & ostr = cout);
    void addWord(const string & str);
    bool isStored(const string & str) const;
  private:
    BinarySearchTree<string> words;
    string Normalize(const string & str) const;
};
```

```
Dictionary::Dictionary()
{
}

void Dictionary::read(istream & istr)
{
    string word;
    while (istr >> word) {
        words.insert(word);
    }
}

void Dictionary::write(ostream & ostr)
{
    words.display(ostr);
}

void Dictionary::addWord(const string & str)
{
    string normal = Normalize(str);

    if (normal != "" && !isStored(normal)) {
        words.insert(normal);
    }
}

bool Dictionary::isStored(const string & word) const
{
    return words.isStored(word);
}
```

would this implementation work well in our speller program?

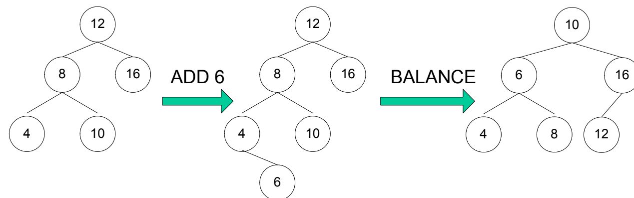
modifications?

Balancing trees

on average, N random insertions into a BST yields O(log N) height

- however, degenerative cases exist (e.g., if data is close to ordered)

we can ensure logarithmic depth by maintaining balance



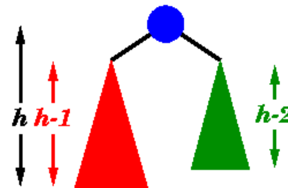
maintaining full balance can be costly (see next HW)

- however, full balance is not needed to ensure O(log N) operations
- specialized structures/algorithms exist: AVL trees, 2-3 trees, red-black trees, …

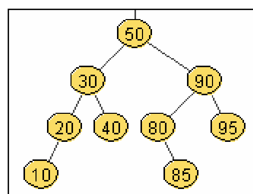
AVL trees

an AVL tree is a binary search tree where

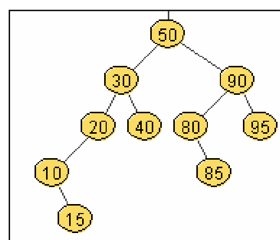
- for every node, the heights of the left and right subtrees differ by at most 1



- first self-balancing binary search tree variant
- named after Adelson-Velskii & Landis (1962)



AVL tree

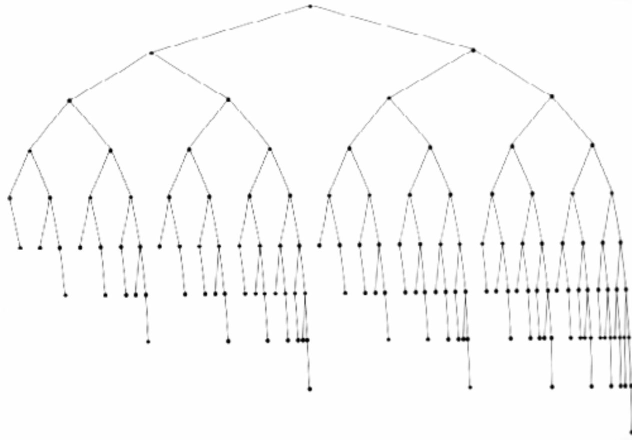


not an AVL tree – WHY?

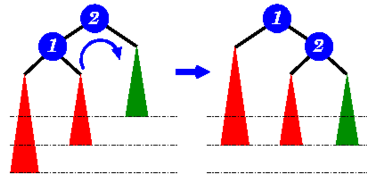
AVL trees and balance

the AVL property is weaker than full balance, but sufficient to ensure logarithmic height

- height of AVL tree with N nodes < 2 log(N) + 2 → searching is O(log N)

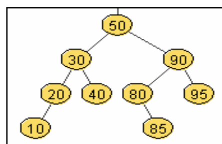


Inserting/removing from AVL tree

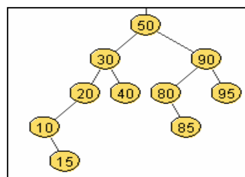


when you insert or remove from an AVL tree, may need to rebalance

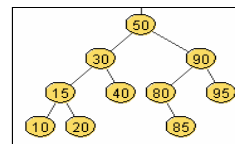
- add/remove value as with binary search trees
- may need to rotate subtrees to rebalance
- see http://www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html



consider AVL tree



inserting ruins balance



move up levels & rotate

worst case, inserting/removing requires traversing the path back to the root and rotating at each level

- each rotation is a constant amount of work → inserting/removing is O(log N)